Universidad Autónoma del Estado de Hidalgo

ICBI

2.4 Análisis sintáctico

AUTOMATAS Y LENGUAJES FORMALES

ALUMNO: JORGE LUIS ORTEGA PÉREZ

Dr. Eduardo Cornejo-Velázquez

3.9. Ejercicios y actividades

1. a) Escriba una gramática que genere el conjunto de cadenas (s; , s;s; , s;s;s; , . . .).

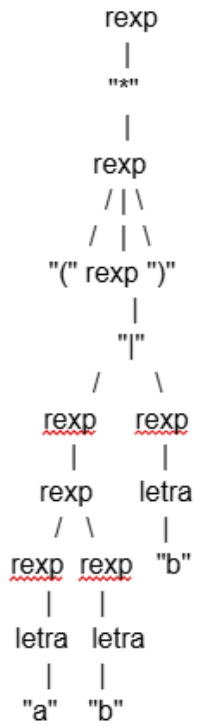
Solución:
Variables (V): S
Terminales (Σ): s,;
Reglas de producción (R):
$S \rightarrow s;$
$S \rightarrow s; S$
Símbolo inicial (S): S

b) Genere un árbol sintáctico para la cadena s;s;

2. Considere la siguiente gramática:

rexp → rexp ”I” rexp
— rexp rexp
— rexp ”*”
— ”(” rexp ”)”
— letra

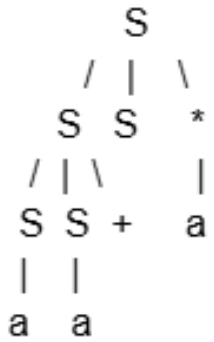
a) Genere un árbol sintáctico para la expresión regular (ab—b)*.



3. De las siguientes gramáticas, describa el lenguaje generado por la gramática y genere árboles sintácticos con las respectivas cadenas.

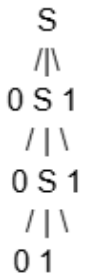
a) S → S S + I S S * I a con la cadena aa+a*.

La gramática genera expresiones postfijas con operadores + y *, donde "a" es un operando. Ejemplos de cadenas válidas: aa+, aaa**, aa+a*, aaa+*.



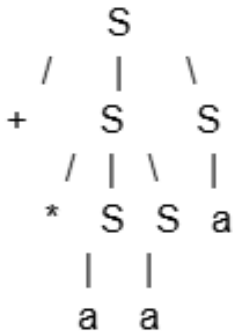
b) S → 0 S 1 I 0 1 con la cadena 000111.

La gramática genera cadenas de la forma 0^n1^n; igual número de ceros seguidos de unos. Ejemplos de cadenas válidas: 01, 0011, 000111, 00001111.



c) S → + S S I * S S I a con la cadena + * aaa.

La gramática genera expresiones prefijas con operadores + y *, donde "a" es un operando. Ejemplos de cadenas válidas: +aa, *aa, +*aaa, **aaa.



4. ¿Cuál es el lenguaje generado por la siguiente gramática?

S → xSy I ϵ

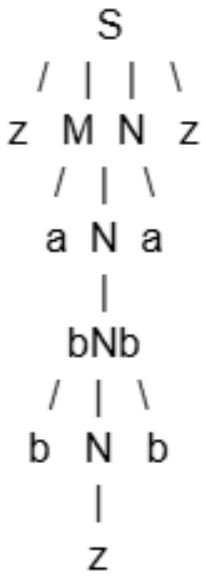
La gramática genera cadenas de la forma $x^n y^n$, donde $n \geq 0$. Esto significa que el número de "x" es igual al número de "y", y puede incluir la cadena vacía (ϵ).

El lenguaje generado es: $L = \{x^n y^n | n \geq 0\}$

El lenguaje consiste en cadenas balanceadas con igual cantidad de x e y, como ϵ, xy, xxyy, xxxyyy, etc. La regla recursiva xSy asegura que cada x tenga su correspondiente y.

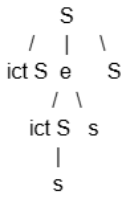
5. Genere el árbol sintáctico para la cadena zazabzbz utilizando la siguiente gramática:

S → zMNz
M → aNa
N → bNb
N → z

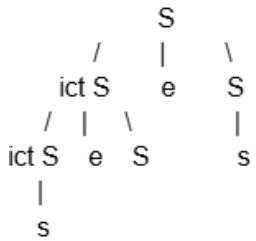


6. Demuestre que la gramática que se presenta a continuación es ambigua, mostrando que la cadena ictictses tiene derivaciones que producen distintos árboles de análisis sintáctico.

S → ictS
S → ictSeS
S → s
Solución: Para demostrar que la gramática es ambigua, debemos encontrar dos árboles de derivación distintos para la cadena.
Primer árbol de derivación: Se aplica S → ictS primero y luego usa S → ictSeS.



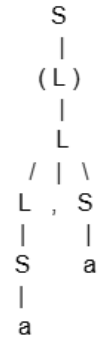
Segundo árbol de derivación: Aquí se aplica primero S → ictSeS y luego S → ictS en una de las expansiones.



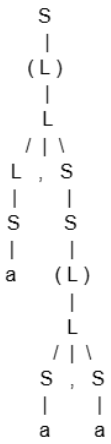
7. Considere la siguiente gramática

S → (L) I a
L → L , S I S

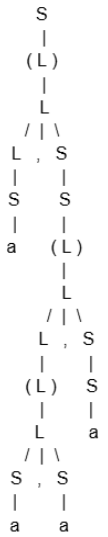
Encuéntrese árboles de análisis sintáctico para las siguientes frases:
a) (a, a)



b) (a, (a, a))

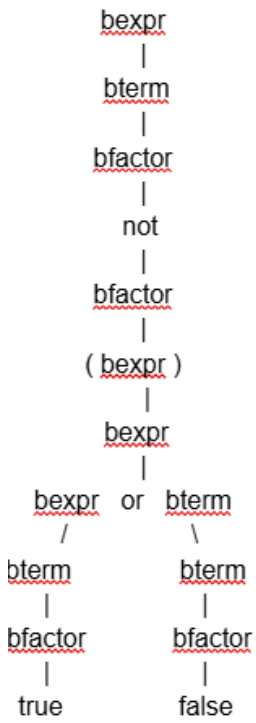


c) (a, ((a, a), (a, a)))



8. Construyase un árbol sintáctico para la frase not (true or false) y la gramática:

bexpr → bexpr or bterm I bterm
bterm → bterm and bfactor I bfactor
bfactor → not bfactor I (bexpr) I true I false



9. Diseñe una gramática para el lenguaje del conjunto de todas las cadenas de símbolos 0 y 1 tales que todo 0 va inmediatamente seguido de al menos un 1

La siguiente gramática es la que define el lenguaje:

$$S \rightarrow 1S \mid 01S \mid \epsilon$$

- La producción $S \rightarrow 1S$ permite generar cualquier cantidad de unos al principio o en medio de la cadena.
- La producción $S \rightarrow 01S$ asegura que cada 0 en la cadena esté seguido inmediatamente por al menos un 1, ya que A→1S genera un 1 y luego permite que la cadena continúe con cualquier cantidad de 1's.

Ejemplos:
La cadena "011" se deriva como:

$$S \rightarrow 01S \rightarrow 011S \rightarrow 011\epsilon \rightarrow 011$$

La cadena "111" se deriva como:

$$S \rightarrow 1S \rightarrow 11S \rightarrow 111S \rightarrow 111\epsilon \rightarrow 111$$

La cadena "0101" se deriva como:

$$S \rightarrow 01S \rightarrow 0101S \rightarrow 0101\epsilon \rightarrow 0101$$

10. Elimine la recursividad por la izquierda de la siguiente gramática:

S → (L) | a
Lß$L, S \mid S$

Donde se encuentra L, contiene una recursividad por la izquierda ubicada en L → L, S, puesto que el símbolo L aparece en el lado izquierdo de su propia producción. El proceso de eliminar la recursividad por la izquierda de una gramática sigue un conjunto de pasos sistemáticos.

- Identificar recursividad por la izquierda: La producción de L tiene recursividad por la izquierda: L→L,S.

- Reescribir las producciones de forma no recursiva: Se introduce un nuevo no terminal L' y se reescribe:
L→SL'
L' →,SL' | ϵ

Resultado final (sin recursividad por la izquierda):
S → $(L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow, SL' \mid \epsilon$

Explicación
- S→(L) | a : $La producción de S no cambia, sigue generando cadenas que comienzan con (L) o con a.$
$-$ LßSL′ : $La producción de L ahora comienza con un S seguido de L′, que manejará el resto de la cadena.$
$-$ L′ß, SL′ | ϵ : $L′ genera una secuencia de$ ”, S”$(lo que permite separar múltiples términos)$
$o termina con (lo que significa que no hay más elementos después de S).$

De esta manera se termina con la recursividad por la izquierda de la gramática original, por lo que se vuelve adecuada para ser procesada por algoritmos de análisis sintáctico que requieran gramáticas sin recursividad por la izquierda.

11. Dada la gramática S → (S) — x, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

```
FUNCIÓN analizar_S():
TOKEN = obtener_siguiente_token()

SI TOKEN == '(' ENTONCES:
    consumir('(')    # Verifica y consume el '('
    analizar_S()    # Llama recursivamente a S
    consumir(')')    # Verifica y consume el ')'

SINO SI TOKEN == 'x' ENTONCES:
    consumir('x')    # Verifica y consume 'x'

SINO:
ERROR("Token inesperado: esperaba '(' o 'x'")
FIN SI
FIN FUNCIÓN
```

12. Qué movimientos realiza un analizador sintáctico predictivo con la entrada (id+id)*id, mediante el algoritmo 3.2, y utilizándose la tabla de análisis sintáctico de la tabla 3.1.

Tabla 3.1
Tabla de análisis sintáctico para la gramática 3.3

No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	E → TE ‘			E → TE ‘		
E ‘		E ‘→ +TE ‘			E ‘ → ε	E ‘ → ε
T	T → FT ‘			T → FT ‘		
T ‘		T ‘ → ε	T ‘→ *FT ‘		T ‘ → ε	T ‘ → ε
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Algoritmo 3.2: Análisis sintáctico predictivo, controlado por una tabla. (Aho, Lam, Sethi, & Ullman, 2008, pág. 226)

Entrada: Una cadena w y una tabla de análisis sintáctico M para la gramática G.

Salida: Si w está en el lenguaje de la gramática $L(G)$, una derivación por la izquierda de w; de lo contrario, una indicación de error.

Método: Al principio, el analizador sintáctico se encuentra en una configuración con w\$ en el búfer de entrada, y el símbolo inicial S de G en la parte superior de la pila, por encima de \$.

```
establecer ip para que apunte al primer símbolo de w;
establecer X con el símbolo de la parte superior de la pila;
while ( X ≠ $ ) { /* mientras la pila no está vacía*/
    if ( X es a ) extraer de la pila y avanzar ip; /* a=símbolo al que apunta ip */
    else if ( X es un terminal ) error( )
        else if ( M[X, a] es una entrada de error ) error( )
            else if ( M[X, a] = X→ Y1Y2... Yk ) {
                enviar de salida la producción X→ Y1Y2... Yk;
                extraer de la pila;
                meter Yk, Yk-1, ..., Y1 en la pila, con Y1 en la parte superior;
                }
    establecer X con el símbolo de la cima de la pila;
}
```

Pila	**Entrada**	**Acción**
\$E	(id+id)*id\$	E → TE’
\$E’ T	(id+id)*id\$	T → FT’
\$E’ T’ F	(id+id)*id\$	F → (E)
\$E’ T’)E(	(id+id)*id\$	Concuerda(()
\$E’ T’)E	id+id)*id\$	E → TE’
\$E’ T’)E’ T	id+id)*id\$	T → FT’
\$E’ T’)E’ T’ F	id+id)*id\$	F → id
\$E’ T’)E’ T’ id	id+id)*id\$	Concuerda(id)
\$E’ T’)E’ T’	+id)*id\$	T’ →
\$E’ T’)E’	+id)*id\$	E’ → +TE’
\$E’ T’)E’ T+	+id)*id\$	Concuerda(+)
\$E’ T’)E’ T	id)*id\$	T → FT’
\$E’ T’)E’ T’ F	id)*id\$	F → id
\$E’ T’)E’ T’ id	id)*id\$	Concuerda(id)
\$E’ T’)E’ T’	)*id\$	T’ →
\$E’ T’)E’	)*id\$	E’ →
\$E’ T’)	)*id\$	Concuerda())
\$E’ T’	*id\$	T’ → *FT’
\$E’ T’ F*	*id\$	Concuerda(*)
\$E’ T’ F	id\$	F → id
\$E’ T’ id	id\$	Concuerda(id)
\$E’ T’	\$	T’ →
\$E’	\$	E’ →
\$	\$	Aceptar

13. La gramática 3.2, sólo maneja las operaciones de suma y multiplicación, modifique esa gramática para que acepte, también, la resta y la división; Posteriormente, elimine la recursividad por la izquierda de la gramática completa y agregue la opción de que F, también pueda derivar en num, es decir, F → (E) — id — num

Gramática Original (Solo Suma y Multiplicación)
La gramática inicial solo soporta + y *:
E $\rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Inclusión de Resta (-) y División (/)
Se añaden las operaciones de resta y división, siguiendo la misma jerarquía que + y *:
E $\rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T/F \mid F$
$F \rightarrow (E) \mid id$

Eliminación de Recursividad por la Izquierda
Para evitar recursividad infinita (ej: E → E + T → E + T + T → ...), se introducen nuevos no terminales E' y T':
Reglas para E (Expresiones):
E $\rightarrow TE'$
$E' \rightarrow +TE' \mid -TE' \mid \epsilon$

Reglas para T (Términos):
T $\rightarrow FT'$
$T' \rightarrow *FT' \mid /FT' \mid \epsilon$

Extensión de F para Incluir Números (num)
Se añade num como alternativa a id y paréntesis:
F → (E) | $id \mid num$

Gramática Final (Sin Recursividad y con Todas las Operaciones)
E → T E'
E' → + T E' | $-TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid /FT' \mid \epsilon$
$F \rightarrow (E) | id | num$

14. Escriba un pseudocódigo (e implemente en Java) utilizando el método descendente recursivo para la gramática resultante del ejercicio anterior (ejercicio 13).

```
FUNCIÓN analizar_E():
   analizar_T()
   analizar_E'()

FUNCIÓN analizar_E'():
```

```
   TOKEN = obtener_siguiente_token()
   SI TOKEN == '+' O TOKEN == '-' ENTONCES:
    consumir(TOKEN)
    analizar_T()
    analizar_E'()
   SINO:
    # : no hacer nada
   FIN SI

FUNCIÓN analizar_T():
   analizar_F()
   analizar_T'()

FUNCIÓN analizar_T'():
   TOKEN = obtener_siguiente_token()
   SI TOKEN == '*' O TOKEN == '/' ENTONCES:
    consumir(TOKEN)
    analizar_F()
    analizar_T'()
   SINO:
    # : no hacer nada
   FIN SI

FUNCIÓN analizar_F():
   TOKEN = obtener_siguiente_token()
   SI TOKEN == '(' ENTONCES:
    consumir('(')
    analizar_E()
    consumir(')')
   SINO SI TOKEN == 'id' O TOKEN == 'num' ENTONCES:
    consumir(TOKEN)
   SINO:
    ERROR("Token inesperado en F")
   FIN SI
```

Implementación en Java:

```
import java.util.*;

public class AnalizadorSintactico {
    private List<String> tokens;
    private int currentTokenIndex;

    public AnalizadorSintactico(List<String> tokens) {
        this.tokens = tokens;
        this.currentTokenIndex = 0;
    }

    private String getNextToken() {
        if (currentTokenIndex < tokens.size()) {
            return tokens.get(currentTokenIndex);
        }
        return "$"; // Fin de entrada
    }
```

```
private void consume(String expectedToken) {
    if (getNextToken().equals(expectedToken)) {
        currentTokenIndex++;
    } else {
        throw new RuntimeException("Error: se esperaba " + expectedToken);
    }
}

public void analizar_E() {
    analizar_T();
    analizar_Eprima();
}

private void analizar_Eprima() {
    String token = getNextToken();
    if (token.equals("+") || token.equals("-")) {
        consume(token);
        analizar_T();
        analizar_Eprima();
    }
    // : no se hace nada
}

private void analizar_T() {
    analizar_F();
    analizar_Tprima();
}

private void analizar_Tprima() {
    String token = getNextToken();
    if (token.equals("*") || token.equals("/")) {
        consume(token);
        analizar_F();
        analizar_Tprima();
    }
    // : no se hace nada
}

private void analizar_F() {
    String token = getNextToken();
    if (token.equals("(")) {
        consume("(");
        analizar_E();
        consume(")");
    } else if (token.equals("id") || token.equals("num")) {
        consume(token);
    } else {
        throw new RuntimeException("Error: token inesperado en F: " + token);
    }
}

public static void main(String[] args) {
    // Ejemplo de uso con la cadena: (id - num) / id
    List<String> tokens = Arrays.asList("(", "id", "-", "num", ")", "/", "id");
```

```
        AnalizadorSintactico analizador = new AnalizadorSintactico(tokens);
        try {
            analizador.analizar_E();
            System.out.println("¡Cadena válida!");
        } catch (RuntimeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```